Improve the VM live migration process

In this study, we'll aim at improving the VM live migration from a user experience perspective. We'll particularly focus on the live migration of VM with non-shared storage (i.e, the source VM and the destination VM do not share the same storage through a NFS technology like `Ceph` for example)

In order to enable the live migration for a VM, the user must specify a `size.state` parameter for the instance root disk. This parameter is used for two things:

- In case of a stateful `stop` of the instance during the migration process, the instance need to save its current memory pages (plus CPU state) to the disk in order to be able to resume its execution after a stateful `start`. This state dump is saved in the root disk config volume, thus justifying the `size.state` parameter being **at least** equal to the amount of memory allocated to the instance in order for the root disk config volume to be able to store the state dump.
- Why do we need '**at least**' the VM memory size and not an '**equal**' amount for our `size.state` parameter? Well, the stored state of the VM also contain other data. Indeed, during the live migration process, we let QEMU mirror the storage device live-writes to a copy-on-write file: a `.qcow2` file. This file is written on the root disk config volume.

The following of this study will propose a strategy describing how we could get rid of the `size.state` parameter and make the live migration process more *adaptive* and user-friendly. In a second study, we'll cover how we can get rid of the `migration.stateful` parameter so that the user does not have to specify any parameter at all to enable the live migration of its VM.

Strategy

Let's have a new shared volume managed by LXD noted Ω with a fixed size Ω_S. This size Ω_S is LXD specific and could be a modifiable server parameter but will come with a chosen default value (e.g, `10GiB`). The purpose of this shared volume is to act as a temporary buffer in order to store the `.qcow2` files of the running VM live migration processes (we could name this volume `migration-cow-buffer`. This volume could be backed by ZFS by default).

I don't think there would be a need to store any VM memory dumps in this volume because each memory dump has a determined upper bound for its size (unlike the live-writes `.qcow2` file), so it is easy to resize the VM root disk at the pre-migration phase to make sure the root disk config volume is big enough to store the memory dump and the CPU state. We'd just need to resize the root disk on the target node to its original size after the migration process is done.

The Ω volume would be partitioned into multiple sub-volumes noted ω_i which have a size $\Pi(\omega_i)$ where Π is a partitioning function. We'll come back to this partitioning function a bit later; for now, let's assume it exist. There would be a 1:1 mapping between the sub-volumes and the VM live migration processes so that each ω_i holds one `.qcow2` file. Naturally, we'd have $\sum_{i=1}^{n} \Pi(\omega_i) = \Omega_S$. We also note $s_{\text{cow},i}(t)$ the size of the `.qcow2` file associated to the ω_i sub-volume at time t. We have $\forall t, s_{\text{cow},i}(t) \leq \Pi(\omega_i)$ (if $s_{\text{cow},i}(t)$ is not 'sharded' - see section `3. 'Bins' overlapping`).

Each ω_i would have an associated **throttle rate** $\gamma_i(t)$. The throttle rate is a variable limit that will be applied to a QEMU storage device through QMP (we can also use the `-drive` option with `throttling.*` parameters to set IOPS limits on the disk. For example: `-drive file=/path/to/disk.img,throttling.bps-write=10485760,...` would limit the write bandwidth to `10 MB/s`.)

It is important to note that the content of an ω_i can start to be consumed only after the disk content prior to the migration has been transferred. The remaining size of this disk prior to the migration is noted $p_i(t)$. Until then, $s_{\text{cow},i}(t)$ is only growing but its size is capped by $\Pi(\omega_i)$ (if not 'sharded') and its growth a time t is capped by $\gamma_i(t)$.

Let's note $U(t)$ the current speed of the consuming link (the link between the source and the destination nodes).

Objective: Our goal is to update the throttle rates $\gamma_i(t)$ of each VM live migration process so that all the `.qcow2` files contained in the partitions of Ω keep a decent IOPS without the total size Ω_S being entirely filled up.

Π : The partitioning function

We need to design an ideal partitioning function, Π, for dynamically managing sub-spaces (or shortly put, 'bins'. We'll use these two terms interchangeably) within a shared buffer Ω for `.qcow2` files during multiple live migrations. This function needs to balance the requirements of each migration while ensuring the total fixed size of Ω_S is not exceeded. Let's define a few principles and then propose a partitioning strategy.

Principles

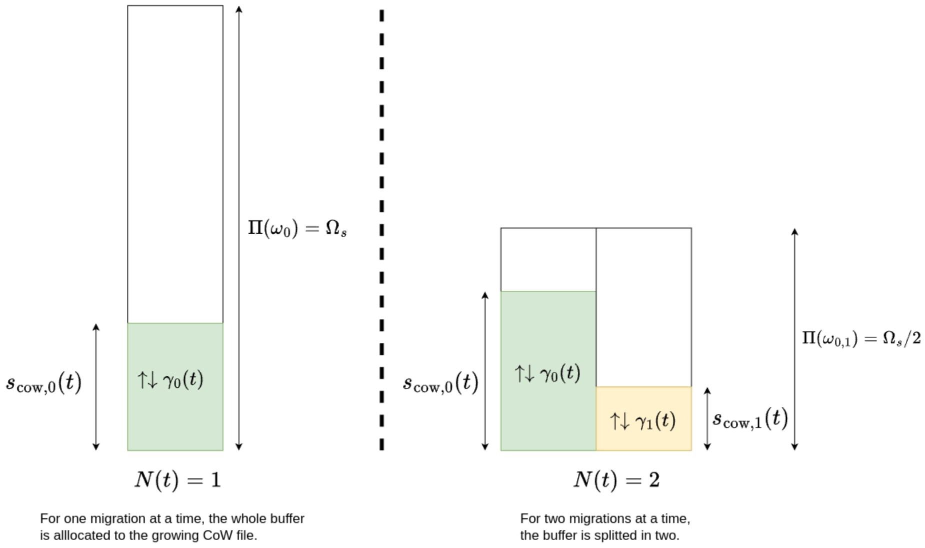
- **Dynamic allocation**: the partitioning function should allocate sub-spaces based on current requirements and available space, adjusting as migrations start and end.
- **Fairness**: each migration process should get a fair share of the buffer, ideally based on its needs and the total available space.
- **Buffer limitation**: the sum of all partitions should never exceed Ω_S.
- **Non-reduction of allocated space**: once a space is allocated to a `.qcow2` file, it can't be reduced (this will lead to data loss / corruption).

Proposed function

Let $N(t)$ be the number of ongoing migrations at time t. The function can be defined as follows:

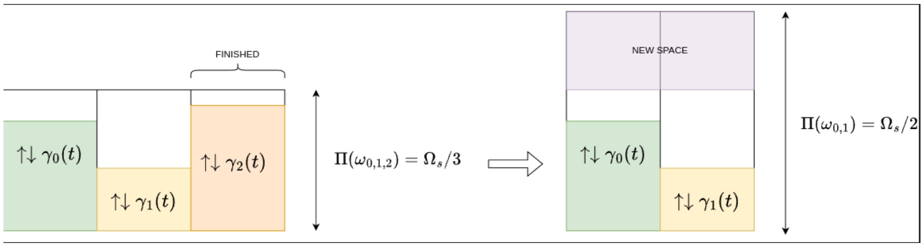
1. **Starting a new migration - initial allocation**:

When a new migration starts, allocate an initial partition size based on the available space and the number of ongoing migrations. $\Pi(\omega_{\text{new}}) = \frac{\Omega_s}{N(t)+1}$


$\Pi(\omega_0) = \Omega_s$
$s_{\text{cow},0}(t)$
$\uparrow\downarrow\ \gamma_0(t)$
$N(t) = 1$
For one migration at a time, the whole buffer is alllocated to the growing CoW file.
$s_{\text{cow},0}(t)$
$\uparrow\downarrow\ \gamma_0(t)$
$\uparrow\downarrow\ \gamma_1(t)$
$s_{\text{cow},1}(t)$
$\Pi(\omega_{0,1}) = \Omega_s/2$
$N(t) = 2$
For two migrations at a time, the buffer is splitted in two.

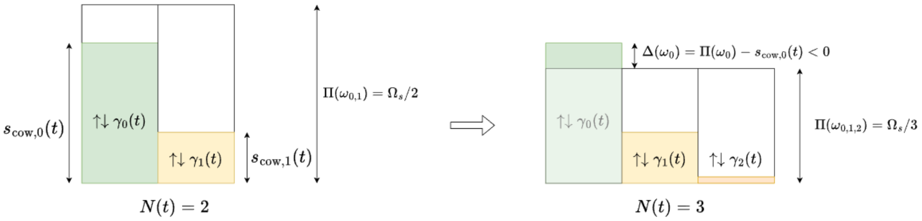
2. **Completed migration - reclaiming space**:

When a migration completes (one of the 'bin' on the diagram above is no longer needed because its underlying `.qcow2` has been removed), the other 'bins' can reclaim the space allocated to the completed migration. The reclaimed space is then redistributed to the remaining ongoing migrations. The redistribution is done by increasing the size of each remaining 'bin' by the same amount. The amount is calculated by dividing the reclaimed space by the number of ongoing migrations:


FINISHED
$\uparrow\downarrow\ \gamma_0(t)$
$\uparrow\downarrow\ \gamma_1(t)$
$\uparrow\downarrow\ \gamma_2(t)$
$\Pi(\omega_{0,1,2}) = \Omega_s/3$
NEW SPACE
$\uparrow\downarrow\ \gamma_0(t)$
$\uparrow\downarrow\ \gamma_1(t)$
$\Pi(\omega_{0,1}) = \Omega_s/2$

3. **'Bins' overlapping**

Now, you are probably wondering what's happening in the case where a new migration causes a bin to be full or to be less than the associated $s_{\text{cow},i}(t)$ (this is because a bin's size is divided when a new migration process occurs). Or, what's happening if a migration is hogging its bin and the other migrations are starving or just simply haven't filled up their respective bins?

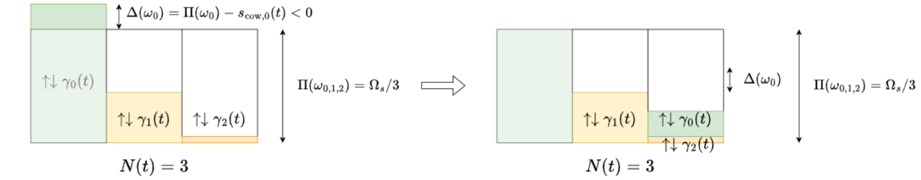

$s_{\text{cow},0}(t)$
$\uparrow\downarrow \gamma_0(t)$
$\uparrow\downarrow \gamma_1(t)$
$s_{\text{cow},1}(t)$
$\Pi(\omega_{0,1}) = \Omega_s/2$
$N(t) = 2$
$\Delta(\omega_0) = \Pi(\omega_0) - s_{\text{cow},0}(t) < 0$
$\uparrow\downarrow \gamma_0(t)$
$\uparrow\downarrow \gamma_1(t)$
$\uparrow\downarrow \gamma_2(t)$
$\Pi(\omega_{0,1,2}) = \Omega_s/3$
$N(t) = 3$

For two migrations at a time, the buffer is splitted in two.

For three migrations at a time, the buffer is splitted in three. **However, this first bin is less than its underlying CoW data.**

In such a situation, we won't just give up and mark migration 0 as failed. **Instead, we'll move the $\Delta(\omega_0)$ bit into a new bin (the least used one in priority) that can withstand the move**. If there is no other bin that can handle the move, we divide $\Delta(\omega_0)$ in 2 and move the $\Delta(\omega_0)/2$ parts in two bins that can handle it. We keep dividing until we can move $\Delta(\omega_0)$. If we still can't we'll mark migration 0 as failed. However, in practice, this should rarely happen because we'd have already set γ_0 to its minimum acceptable level (defined by our SLAs. For example, we could set it to `10 kB/s`).

Then the bin will be marked as full and no underlying data will be grown inside it (until a migration process finishes and some space is reclaimed).


$\Delta(\omega_0) = \Pi(\omega_0) - s_{\text{cow},0}(t) < 0$
$\uparrow\downarrow \gamma_0(t)$
$\uparrow\downarrow \gamma_1(t)$
$\uparrow\downarrow \gamma_2(t)$
$\Pi(\omega_{0,1,2}) = \Omega_s/3$
$N(t) = 3$
$\Delta(\omega_0)$
$\uparrow\downarrow \gamma_1(t)$
$\uparrow\downarrow \gamma_0(t)$
$\uparrow\downarrow \gamma_2(t)$
$\Pi(\omega_{0,1,2}) = \Omega_s/3$
$N(t) = 3$

For three migrations at a time, the buffer is splitted in three. **However, this first bin is less than its underlying CoW data.**

For three migrations at a time, the buffer is splitted in three. **However, this first bin is less than its underlying CoW data.**

With this heuristic, we now have potentially bins holding different `.qcow2` with different γ_i. Let's say that a the j-th bin holds two different type of data each with a different throttling rate, noted γ_i and γ_j. Let's note $\gamma_{i,j}(t) = \gamma_i(t) + \gamma_j(t)$. Because we are in the j-th bin, we chose to give a higher priority to the j-th migration process, so that we avoid the i-th migration process to hog the resources of other bins. This translates by having $\forall t, \gamma_{\text{SLA}} \leq \gamma_i(t) \leq \gamma_j(t)$.

(How the different throttle rates will be adjusted is explained in the next section.)

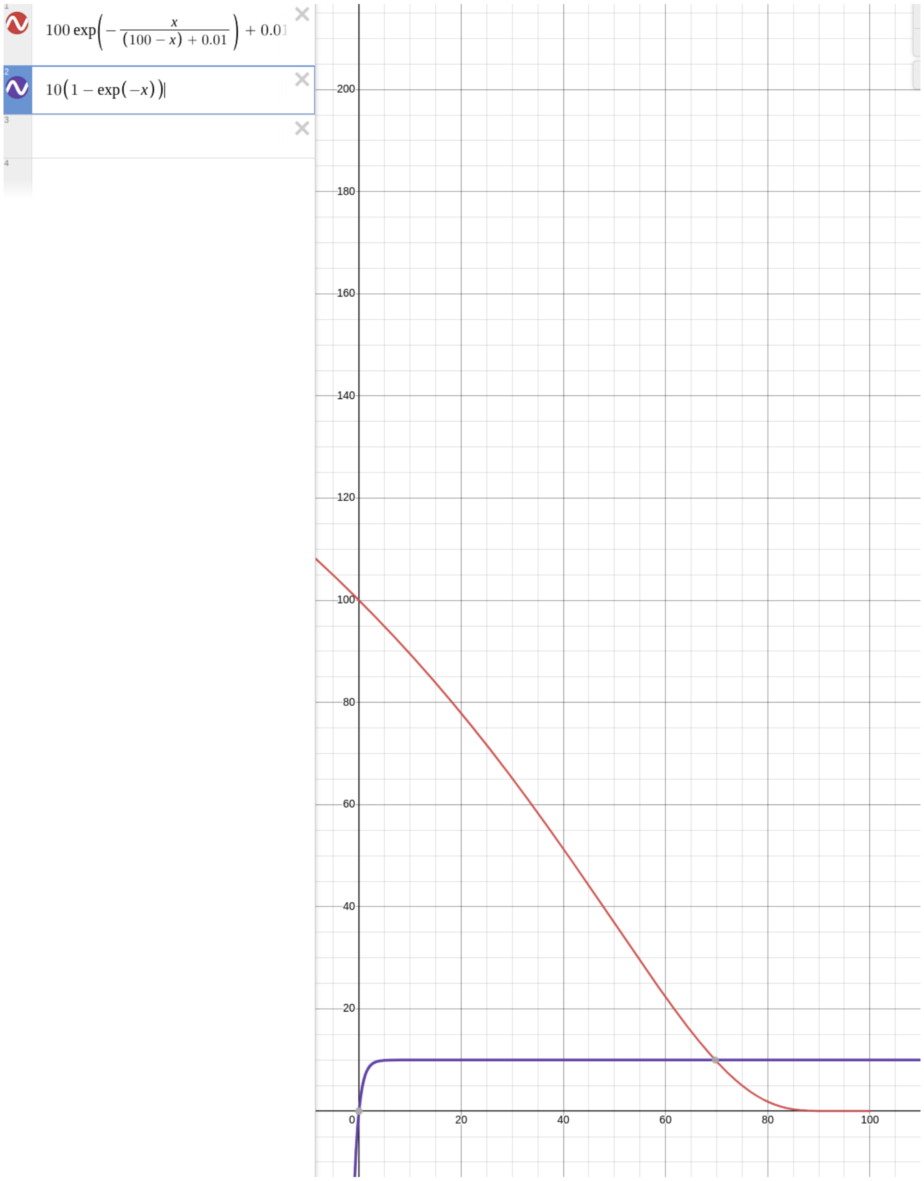
During a completed migration phase, the reclaimed space is redistributed as usual (detailed in step 2), but if a bin is 'sharded' into k parts (i.e, `.qcow2` data for a migration spans across multiple bins), we try to move the $(k-1)$-smallest shards into its original bin. If we can't, we try to move the $(k-2)$-smallest shards into the original bin. We continue to diminish the number of to-be-moved shards until we can do the move. If we still can't, mark the migration (the 'sharded' one) as failed.

γ_i : the throttle rate (non-sharded behavior)

- For a **non-sharded** migration i, we propose the following heuristic to adjust the throttle rate $\gamma_i(t)$:

$\gamma_i(t) = \min(\Pi(\omega_i)\exp(-\frac{s_{\text{cow},i}(t)}{(\Pi(\omega_i) - s_{\text{cow},i}(t)) + \epsilon}) + \epsilon, \frac{U(t)}{N(t)})$

Let's explain the terms and the underlying idea behind this heuristic alongside the following picture:



$100\exp\left(-\frac{x}{(100-x)+0.01}\right)+0.01$

$10(1-\exp(-x))$

The red curve is modeling the left part of the min(.) operator and the blue curve is modeling the right part.

- The blue curve is a simple, yet in practice, close to the reality model of a network bandwidth (a transitive state followed by a steady state) divided by $N(t)$ which is constant most of the time (it is a ‘stair’ function to be exact).
- The red curve show the evolution of the throttling rate with $s_{\text{cow},i}(t)$ in the

x-axis. At first, the throttling rate is around the value of $\Pi(\omega_i)$ which is, in this case, above the allocated network bandwidth $U(t)/N(t)$ allocated for this bin. Then we use the value of the blue curve to adjust the throttling rate. The more the bin is filling up, the closest the red curve get to the blue curve and eventually, exponential throttling rate will be applied without reaching 0 because of $\epsilon = 0.01$ (in MB/s).

So if the network bandwidth is large, the throttling rate will very quickly follow the red curve law and will start to decrease in a somewhat linear way until the late exponential behavior appears at around 60% of the bin size (`auto-convergence` behavior).

If the network bandwidth is small, we won't be able to transfer the `.qcow2` data at a fast pace anyway so the throttle rate will be in phase with the blue curve.

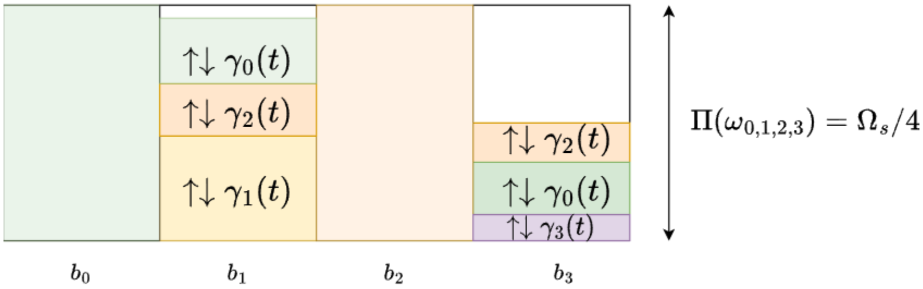
The interesting fact about that heuristic is that this behavior happens for whatever the size of the bin (in the above picture, the bin size for this migration is `100MB/s`) because of the exponential term.

γ_i : the throttle rate (sharded behavior)

As explained in the `partitioning function` section, we can have a migration process that is 'sharded' across multiple bins. We also spoke about some **priorities** that some migrations have over others in the case of a bin being 'sharded'. Let's now explain how these **priorities** are expressed and explain how we can adjust the throttle rate of a migration process that is 'sharded' across multiple bins.

The chosen heuristic is surprisingly simple, because it is based on the same principle as the non-sharded behavior though it contains a simple addition:

Let's picture this situation:



$\uparrow\downarrow \gamma_0(t)$
$\uparrow\downarrow \gamma_2(t)$
$\uparrow\downarrow \gamma_1(t)$
$\uparrow\downarrow \gamma_2(t)$
$\uparrow\downarrow \gamma_0(t)$
$\uparrow\downarrow \gamma_3(t)$
$\Pi(\omega_{0,1,2,3}) = \Omega_s/4$
b_0
b_1
b_2
b_3

We have 4 concurrent migration. In this scenario, because each migration evolved differently, this translated to having 4 bins, among 2 of them are frozen.

- Migration 0 is sharded across 3 bins: its 'main' bin b_0 (all the migrations have at least one shard) and on b_1 and b_3.

- Migration 2 is also sharded across 3 bins: its 'main' bin b_2 and on b_1 and b_3.

You notice that the order of the shards is not the same for both migrations: In b_1, the green shard is on top of the orange one and on b_3, the orange shard is on top of the green one. **Each time a shard is added to a bin, it is added on top of the other shards. The bottom shard is the one with the highest priority and then the order of priority is from bottom to top**.

We introduce the integer $\alpha_i \in [1..N(t)]$, the **priority** of the migration i. The closer α_i is to 1, the higher the priority of the migration i is.

- In b_1, $\alpha_0 = 3$ and $\alpha_2 = 2$
- In b_3, $\alpha_0 = 2$ and $\alpha_2 = 3$

We note $\Sigma\alpha_i$, the **mean priority** of the migration process over the bins it is sharded on (noted B) (including its 'main' bin):

$$\Sigma\alpha_i = \frac{\sum_{j \in B} \alpha_i(b_j)}{\mathrm{card}(B)}$$

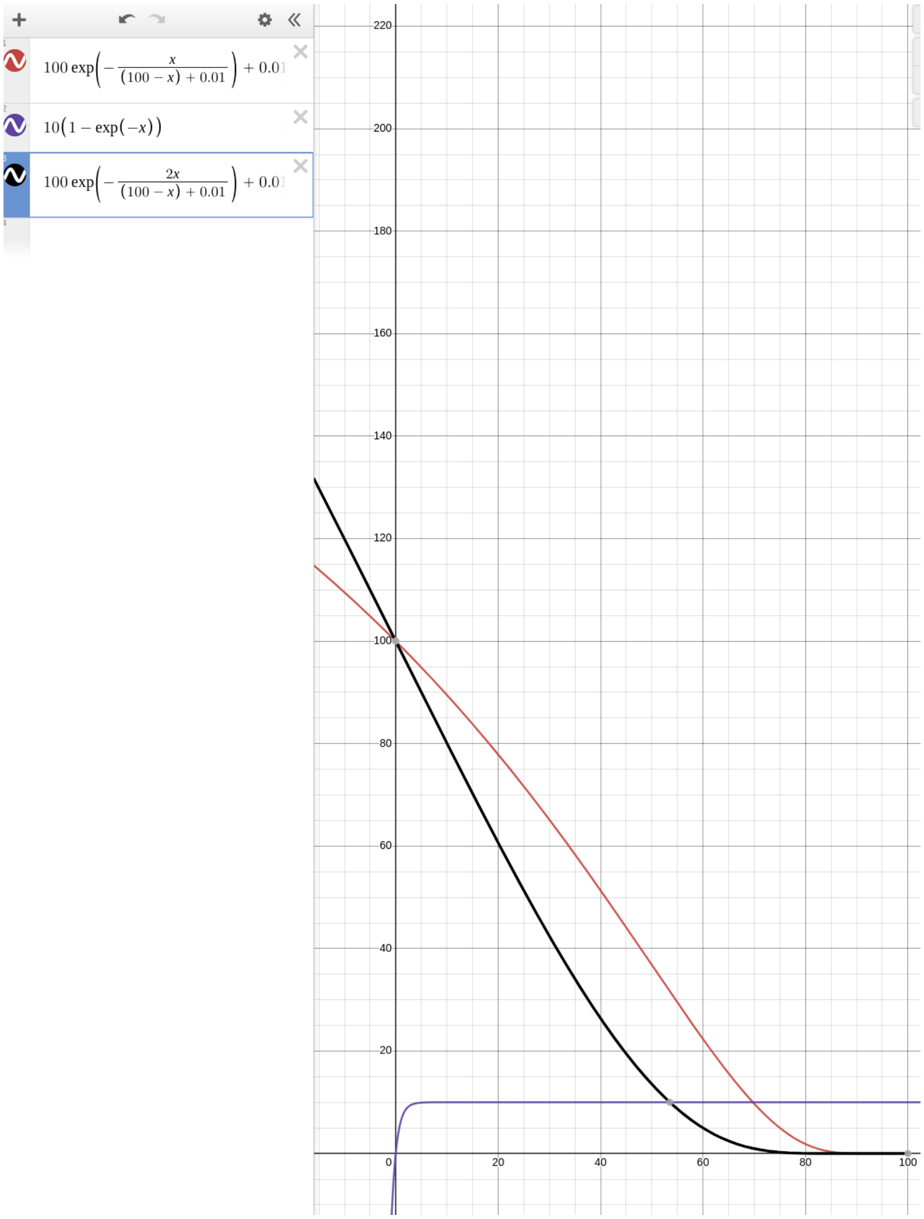
Here, $\Sigma\alpha_0 = \frac{1+2+3}{3} = 2$ and $\Sigma\alpha_2 = \frac{1+2+3}{3} = 2$.

In this end, they have both the same mean priority. Now, this **mean priority** will penalize its γ_i like the following:

$$\gamma_i(t) = \min(\Pi(\omega_i)\exp(-\frac{\Sigma\alpha_i \times s_{\mathrm{cow},i}(t)}{(\Pi(\omega_i) - s_{\mathrm{cow},i}(t)) + \epsilon}) + \epsilon, \frac{U(t)}{N(t)})$$

As you can see, this is consistent with the former definition for a non-sharded migration: indeed, for a non-sharded migration, $\Sigma\alpha_i = 1$ and we fall back on our feet. Wonderful!

Here is what a penalized γ_i looks like when it is sharded with a mean priority of $\Sigma\alpha_i = 2$:



$100\exp\left(-\frac{x}{(100-x)+0.01}\right)+0.0$

$10\left(1-\exp(-x)\right)$

$100\exp\left(-\frac{2x}{(100-x)+0.01}\right)+0.0$

- The black curve is the 'penalized' version of the red curve (the non-sharded behavior). We can clearly see that the effect on the throttling rate is really noticeable for $\Sigma\alpha_i = 2$: after around 40% of the capacity of the sharded bins (the non-frozen ones), the throttling rate has been halved compared to the non-sharded behavior, letting an other migration whose the bin is the main one, more room to fill it up.

Conclusion

Through the monitoring of the bytes being written on the `.qcow2` files of concurrent live VM migrations and a live measurement of the network bandwidth between the source and its destination nodes, we can adjust the IOPS throttle rate of a VM's disk devices. This buffer technology is network aware and adaptive to the host fixed resources while giving a fair share of live-write IOPS per concurrent migration.

It'll allow a LXD server to manage a centralized, hidden volume of a fixed size (decided during the `lxd init` phase, or set by default to `10GiB` with a possibility to be increased / shrink by an admin) and to allow this scheduling logic to automatically throttle any number of concurrent live migrations without the user having to specify any `size.state` parameter.